KEYWORDS

NAME

chan — Read, write and manipulate channels

SYNOPSIS

chan *option* ?*arg arg ...*?

DESCRIPTION

This command provides several operations for reading from, writing to and otherwise manipulating open channels (such as have been created with the **open** and **socket** commands, or the default named channels **stdin**, **stdout** or **stderr** which correspond to the process's standard input, output and error streams respectively). *Option* indicates what to do with the channel; any unique abbreviation for *option* is acceptable. Valid options are:

chan blocked *channelId*

This tests whether the last input operation on the channel called *channelId* failed because it would have otherwise caused the process to block, and returns 1 if that was the case. It returns 0 otherwise. Note that this only ever returns 1 when the channel has been configured to be non-blocking; all Tcl channels have blocking turned on by default.

chan close *channelId* ?*direction*?

Close and destroy the channel called *channelId*. Note that this deletes all existing file-events registered on the channel. If the *direction* argument (which must be **read** or **write** or any unique abbreviation of them) is present, the channel will only be half-closed, so that it can go from being read-write to write-only or read-only respectively. If a read-only channel is closed for reading, it is the same as if the channel is fully closed, and respectively similar for write-only channels. Without the *direction* argument, the channel is closed for both reading and writing (but only if those directions are currently open). It is an error to close a read-only channel for writing, or a write-only channel for reading.

As part of closing the channel, all buffered output is flushed to the channel's output device (only if the channel is ceasing to be writable), any buffered input is discarded (only if the channel is ceasing to be readable), the underlying operating system resource is closed and *channelId* becomes unavailable for future use (both only if the channel is being completely closed).

If the channel is blocking and the channel is ceasing to be writable, the command does not return until all output is flushed. If the channel is non-blocking and there is unflushed output, the channel remains open and the command returns immediately; output will be flushed in the background and the channel will be closed when all the flushing is complete.

If *channelId* is a blocking channel for a command pipeline then **chan close** waits for the child processes to complete.

If the channel is shared between interpreters, then **chan close** makes *channelId* unavailable in the invoking interpreter but has no other effect until all of the sharing interpreters have closed the channel. When the last interpreter in which the channel is registered invokes **chan close** (or **close**), the cleanup actions described above occur. With half-closing, the half-close of the channel only applies to the current interpreter's view of the channel until all channels have closed it in that direction (or completely). See the **interp** command for a description of channel sharing.

Channels are automatically fully closed when an interpreter is destroyed and when the process exits. Channels are switched to blocking mode, to ensure that all output is correctly flushed before the process exits.

The command returns an empty string, and may generate an error if an error occurs while flushing output. If a command in a command pipeline created with **open** returns an error, **chan close** generates an error (similar to the **exec** command.)

Note that half-closes of sockets and command pipelines can have important side effects because they result in a shutdown() or close() of the underlying system resource, which can change how other processes or systems respond to the Tcl program.

chan configure *channelId* ?*optionName*? ?*value*? ?*optionName value*?...

Query or set the configuration options of the channel named *channelId*.

If no *optionName* or *value* arguments are supplied, the command returns a list containing alternating option names and values for the channel. If *optionName* is supplied but no *value* then the command returns the current value of the given option. If one or more pairs of *optionName* and *value* are supplied, the command sets each of the named options to the corresponding *value*; in this case the return value is an empty string.

The options described below are supported for all channels. In addition, each channel type may add options that only it supports. See the manual entry for the command that creates each type of channel for the options supported by that

specific type of channel. For example, see the manual entry for the **socket** command for additional options for sockets, and the **open** command for additional options for serial devices.

-blocking *boolean*
: The **-blocking** option determines whether I/O operations on the channel can cause the process to block indefinitely. The value of the option must be a proper boolean value. Channels are normally in blocking mode; if a channel is placed into non-blocking mode it will affect the operation of the **chan gets**, **chan read**, **chan puts**, **chan flush**, and **chan close** commands; see the documentation for those commands for details. For non-blocking mode to work correctly, the application must be using the Tcl event loop (e.g. by calling **Tcl_DoOneEvent** or invoking the **vwait** command).

-buffering *newValue*
: If *newValue* is **full** then the I/O system will buffer output until its internal buffer is full or until the **chan flush** command is invoked. If *newValue* is **line**, then the I/O system will automatically flush output for the channel whenever a newline character is output. If *newValue* is **none**, the I/O system will flush automatically after every output operation. The default is for **-buffering** to be set to **full** except for channels that connect to terminal-like devices; for these channels the initial setting is **line**. Additionally, **stdin** and **stdout** are initially set to **line**, and **stderr** is set to **none**.

-buffersize *newSize*
: *Newvalue* must be an integer; its value is used to set the size of buffers, in bytes, subsequently allocated for this channel to store input or output. *Newvalue* must be a number of no more than one million, allowing buffers of up to one million bytes in size.

-encoding *name*
: This option is used to specify the encoding of the channel as one of the named encodings returned by **encoding names** or the special value **binary**, so that the data can be converted to and from Unicode for use in Tcl. For instance, in order for Tcl to read characters from a Japanese file in **shiftjis** and properly process and display the contents, the encoding would be set to **shiftjis**. Thereafter, when reading from the channel, the bytes in the Japanese file would be converted to Unicode as they are read. Writing is also supported - as Tcl strings are written to the channel they will automatically be converted to the specified encoding on output.

  If a file contains pure binary data (for instance, a JPEG image), the encoding for the channel should be configured to be **binary**. Tcl will then assign no interpretation to the data in the file and simply read or write raw bytes. The Tcl **binary** command can be used to manipulate this byte-oriented data. It is usually better to set the **-translation** option to **binary** when you want to transfer binary data, as this turns off the other automatic interpretations of the bytes in the stream as well.

  The default encoding for newly opened channels is the same platform- and locale-dependent system encoding used for interfacing with the operating system, as returned by **encoding system**.

-eofchar *char*

-eofchar {*inChar outChar***}**
: This option supports DOS file systems that use Control-z (\x1A) as an end of file marker. If *char* is not an empty string, then this character signals end-of-file when it is encountered during input. For output, the end-of-file character is output when the channel is closed. If *char* is the empty string, then there is no special end of file character marker. For read-write channels, a two-element list specifies the end of file marker for input and output, respectively. As a convenience, when setting the end-of-file character for a read-write channel you can specify a single value that will apply to both reading and writing. When querying the end-of-file character of a read-write channel, a two-element list will always be returned. The default value for **-eofchar** is the empty string in all cases except for files under Windows. In that case the **-eofchar** is Control-z (\x1A) for reading and the empty string for writing. The acceptable range for **-eofchar** values is \x01 - \x7f; attempting to set **-eofchar** to a value outside of this range will generate an error.

-translation *mode*

-translation {*inMode outMode***}**
: In Tcl scripts the end of a line is always represented using a single newline character (\n). However, in actual files and devices the end of a line may be represented differently on different platforms, or even for different devices on the same platform. For example, under UNIX newlines are used in files, whereas carriage-return-linefeed sequences are normally used in network connections. On input (i.e., with **chan gets** and **chan read**) the Tcl I/O system automatically translates the external end-of-line representation into newline characters. Upon output (i.e., with **chan puts**), the I/O system translates newlines to the external end-of-line representation. The default translation mode, **auto**, handles all the common cases automatically, but the **-translation** option provides explicit control over the end of line translations.

The value associated with **-translation** is a single item for read-only and write-only channels. The value is a two-element list for read-write channels; the read translation mode is the first element of the list, and the write translation mode is the second element. As a convenience, when setting the translation mode for a read-write channel you can specify a single value that will apply to both reading and writing. When querying the translation mode of a read-write channel, a two-element list will always be returned. The following values are currently supported:

auto
: As the input translation mode, **auto** treats any of newline (**lf**), carriage return (**cr**), or carriage return followed by a newline (**crlf**) as the end of line representation. The end of line representation can even change from line-to-line, and all cases are translated to a newline. As the output translation mode, **auto** chooses a platform specific representation; for sockets on all platforms Tcl chooses **crlf**, for all Unix flavors, it chooses **lf**, and for the various flavors of Windows it chooses **crlf**. The default setting for **-translation** is **auto** for both input and output.

binary
: No end-of-line translations are performed. This is nearly identical to **lf** mode, except that in addition **binary** mode also sets the end-of-file character to the empty string (which disables it) and sets the encoding to **binary** (which disables encoding filtering). See the description of **-eofchar** and **-encoding** for more information.

cr
: The end of a line in the underlying file or device is represented by a single carriage return character. As the input translation mode, **cr** mode converts carriage returns to newline characters. As the output translation mode, **cr** mode translates newline characters to carriage returns.

crlf
: The end of a line in the underlying file or device is represented by a carriage return character followed by a linefeed character. As the input translation mode, **crlf** mode converts carriage-return-linefeed sequences to newline characters. As the output translation mode, **crlf** mode translates newline characters to carriage-return-linefeed sequences. This mode is typically used on Windows platforms and for network connections.

lf
: The end of a line in the underlying file or device is represented by a single newline (linefeed) character. In this mode no translations occur during either input or output. This mode is typically used on UNIX platforms.

chan copy *inputChan outputChan* ?**-size** *size*? ?**-command** *callback*?
: Copy data from the channel *inputChan*, which must have been opened for reading, to the channel *outputChan*, which must have been opened for writing. The **chan copy** command leverages the buffering in the Tcl I/O system to avoid extra copies and to avoid buffering too much data in main memory when copying large files to slow destinations like network sockets.

The **chan copy** command transfers data from *inputChan* until end of file or *size* bytes or characters have been transferred; *size* is in bytes if the two channels are using the same encoding, and is in characters otherwise. If no **-size** argument is given, then the copy goes until end of file. All the data read from *inputChan* is copied to *outputChan*. Without the **-command** option, **chan copy** blocks until the copy is complete and returns the number of bytes or characters (using the same rules as for the **-size** option) written to *outputChan*.

The **-command** argument makes **chan copy** work in the background. In this case it returns immediately and the *callback* is invoked later when the copy completes. The *callback* is called with one or two additional arguments that indicates how many bytes were written to *outputChan*. If an error occurred during the background copy, the second argument is the error string associated with the error. With a background copy, it is not necessary to put *inputChan* or *outputChan* into non-blocking mode; the **chan copy** command takes care of that automatically. However, it is necessary to enter the event loop by using the **vwait** command or by using Tk.

You are not allowed to do other I/O operations with *inputChan* or *outputChan* during a background **chan copy**. If either *inputChan* or *outputChan* get closed while the copy is in progress, the current copy is stopped and the command callback is *not* made. If *inputChan* is closed, then all data already queued for *outputChan* is written out.

Note that *inputChan* can become readable during a background copy. You should turn off any **chan event** or **fileevent** handlers during a background copy so those handlers do not interfere with the copy. Any I/O attempted by a **chan event** or **fileevent** handler will get a “channel busy” error.

Chan copy translates end-of-line sequences in *inputChan* and *outputChan* according to the **-translation** option for these channels (see **chan configure** above). The translations mean that the number of bytes read from *inputChan* can be different than the number of bytes written to *outputChan*. Only the number of bytes written to *outputChan* is reported, either as the return value of a synchronous **chan copy** or as the argument to the callback for an asynchronous **chan copy**.

Chan copy obeys the encodings and character translations configured for the channels. This means that the incoming

characters are converted internally first UTF-8 and then into the encoding of the channel **chan copy** writes to (see **chan configure** above for details on the **-encoding** and **-translation** options). No conversion is done if both channels are set to encoding **binary** and have matching translations. If only the output channel is set to encoding **binary** the system will write the internal UTF-8 representation of the incoming characters. If only the input channel is set to encoding **binary** the system will assume that the incoming bytes are valid UTF-8 characters and convert them according to the output encoding. The behaviour of the system for bytes which are not valid UTF-8 characters is undefined in this case.

chan create *mode cmdPrefix*

This subcommand creates a new script level channel using the command prefix *cmdPrefix* as its handler. Any such channel is called a **reflected** channel. The specified command prefix, **cmdPrefix**, must be a non-empty list, and should provide the API described in the **refchan** manual page. The handle of the new channel is returned as the result of the **chan create** command, and the channel is open. Use either **close** or **chan close** to remove the channel.

The argument *mode* specifies if the new channel is opened for reading, writing, or both. It has to be a list containing any of the strings "**read**" or "**write**". The list must have at least one element, as a channel you can neither write to nor read from makes no sense. The handler command for the new channel must support the chosen mode, or an error is thrown.

The command prefix is executed in the global namespace, at the top of call stack, following the appending of arguments as described in the **refchan** manual page. Command resolution happens at the time of the call. Renaming the command, or destroying it means that the next call of a handler method may fail, causing the channel command invoking the handler to fail as well. Depending on the subcommand being invoked, the error message may not be able to explain the reason for that failure.

Every channel created with this subcommand knows which interpreter it was created in, and only ever executes its handler command in that interpreter, even if the channel was shared with and/or was moved into a different interpreter. Each reflected channel also knows the thread it was created in, and executes its handler command only in that thread, even if the channel was moved into a different thread. To this end all invocations of the handler are forwarded to the original thread by posting special events to it. This means that the original thread (i.e. the thread that executed the **chan create** command) must have an active event loop, i.e. it must be able to process such events. Otherwise the thread sending them will *block indefinitely*. Deadlock may occur.

Note that this permits the creation of a channel whose two endpoints live in two different threads, providing a stream-oriented bridge between these threads. In other words, we can provide a way for regular stream communication between threads instead of having to send commands.

When a thread or interpreter is deleted, all channels created with this subcommand and using this thread/interpreter as their computing base are deleted as well, in all interpreters they have been shared with or moved into, and in whatever thread they have been transferred to. While this pulls the rug out under the other thread(s) and/or interpreter(s), this cannot be avoided. Trying to use such a channel will cause the generation of a regular error about unknown channel handles.

This subcommand is **safe** and made accessible to safe interpreters. While it arranges for the execution of arbitrary Tcl code the system also makes sure that the code is always executed within the safe interpreter.

chan eof *channelId*

Test whether the last input operation on the channel called *channelId* failed because the end of the data stream was reached, returning 1 if end-of-file was reached, and 0 otherwise.

chan event *channelId event* ?*script*?

Arrange for the Tcl script *script* to be installed as a *file event handler* to be called whenever the channel called *channelId* enters the state described by *event* (which must be either **readable** or **writable**); only one such handler may be installed per event per channel at a time. If *script* is the empty string, the current handler is deleted (this also happens if the channel is closed or the interpreter deleted). If *script* is omitted, the currently installed script is returned (or an empty string if no such handler is installed). The callback is only performed if the event loop is being serviced (e.g. via **vwait** or **update**).

A file event handler is a binding between a channel and a script, such that the script is evaluated whenever the channel becomes readable or writable. File event handlers are most commonly used to allow data to be received from another process on an event-driven basis, so that the receiver can continue to interact with the user or with other channels while waiting for the data to arrive. If an application invokes **chan gets** or **chan read** on a blocking channel when there is no input data available, the process will block; until the input data arrives, it will not be able to service other events, so it will appear to the user to "freeze up". With **chan event**, the process can tell when data is present and only invoke **chan gets** or **chan read** when they will not block.

A channel is considered to be readable if there is unread data available on the underlying device. A channel is also considered to be readable if there is unread data in an input buffer, except in the special case where the most recent

attempt to read from the channel was a **chan gets** call that could not find a complete line in the input buffer. This feature allows a file to be read a line at a time in non-blocking mode using events. A channel is also considered to be readable if an end of file or error condition is present on the underlying file or device. It is important for *script* to check for these conditions and handle them appropriately; for example, if there is no special check for end of file, an infinite loop may occur where *script* reads no data, returns, and is immediately invoked again.

A channel is considered to be writable if at least one byte of data can be written to the underlying file or device without blocking, or if an error condition is present on the underlying file or device. Note that client sockets opened in asynchronous mode become writable when they become connected or if the connection fails.

Event-driven I/O works best for channels that have been placed into non-blocking mode with the **chan configure** command. In blocking mode, a **chan puts** command may block if you give it more data than the underlying file or device can accept, and a **chan gets** or **chan read** command will block if you attempt to read more data than is ready; no events will be processed while the commands block. In non-blocking mode **chan puts**, **chan read**, and **chan gets** never block.

The script for a file event is executed at global level (outside the context of any Tcl procedure) in the interpreter in which the **chan event** command was invoked. If an error occurs while executing the script then the command registered with **interp bgerror** is used to report the error. In addition, the file event handler is deleted if it ever returns an error; this is done in order to prevent infinite loops due to buggy handlers.

chan flush *channelId*
Ensures that all pending output for the channel called *channelId* is written.

If the channel is in blocking mode the command does not return until all the buffered output has been flushed to the channel. If the channel is in non-blocking mode, the command may return before all buffered output has been flushed; the remainder will be flushed in the background as fast as the underlying file or device is able to absorb it.

chan gets *channelId* ?*varName*?
Reads the next line from the channel called *channelId*. If *varName* is not specified, the result of the command will be the line that has been read (without a trailing newline character) or an empty string upon end-of-file or, in non-blocking mode, if the data available is exhausted. If *varName* is specified, the line that has been read will be written to the variable called *varName* and result will be the number of characters that have been read or -1 if end-of-file was reached or, in non-blocking mode, if the data available is exhausted.

If an end-of-file occurs while part way through reading a line, the partial line will be returned (or written into *varName*). When *varName* is not specified, the end-of-file case can be distinguished from an empty line using the **chan eof** command, and the partial-line-but-non-blocking case can be distinguished with the **chan blocked** command.

chan names ?*pattern*?
Produces a list of all channel names. If *pattern* is specified, only those channel names that match it (according to the rules of **string match**) will be returned.

chan pending *mode channelId*
Depending on whether *mode* is **input** or **output**, returns the number of bytes of input or output (respectively) currently buffered internally for *channelId* (especially useful in a readable event callback to impose application-specific limits on input line lengths to avoid a potential denial-of-service attack where a hostile user crafts an extremely long line that exceeds the available memory to buffer it). Returns -1 if the channel was not opened for the mode in question.

chan pipe
Creates a standalone pipe whose read- and write-side channels are returned as a 2-element list, the first element being the read side and the second the write side. Can be useful e.g. to redirect separately **stderr** and **stdout** from a subprocess. To do this, spawn with "2>@" or ">@" redirection operators onto the write side of a pipe, and then immediately close it in the parent. This is necessary to get an EOF on the read side once the child has exited or otherwise closed its output.

Note that the pipe buffering semantics can vary at the operating system level substantially; it is not safe to assume that a write performed on the output side of the pipe will appear instantly to the input side. This is a fundamental difference and Tcl cannot conceal it. The overall stream semantics *are* compatible, so blocking reads and writes will not see most of the differences, but the details of what exactly gets written when are not. This is most likely to show up when using pipelines for testing; care should be taken to ensure that deadlocks do not occur and that potential short reads are allowed for.

chan pop *channelId*
Removes the topmost transformation from the channel *channelId*, if there is any. If there are no transformations added to *channelId*, this is equivalent to **chan close** of that channel. The result is normally the empty string, but can be an error in some situations (i.e. where the underlying system stream is closed and that results in an error).

chan postevent *channelId eventSpec*

This subcommand is used by command handlers specified with **chan create**. It notifies the channel represented by the handle *channelId* that the event(s) listed in the *eventSpec* have occurred. The argument has to be a list containing any of the strings **read** and **write**. The list must contain at least one element as it does not make sense to invoke the command if there are no events to post.

Note that this subcommand can only be used with channel handles that were created/opened by **chan create**. All other channels will cause this subcommand to report an error.

As only the Tcl level of a channel, i.e. its command handler, should post events to it we also restrict the usage of this command to the interpreter that created the channel. In other words, posting events to a reflected channel from an interpreter that does not contain it's implementation is not allowed. Attempting to post an event from any other interpreter will cause this subcommand to report an error.

Another restriction is that it is not possible to post events that the I/O core has not registered an interest in. Trying to do so will cause the method to throw an error. See the command handler method **watch** described in **refchan**, the document specifying the API of command handlers for reflected channels.

This command is **safe** and made accessible to safe interpreters. It can trigger the execution of **chan event** handlers, whether in the current interpreter or in other interpreters or other threads, even where the event is posted from a safe interpreter and listened for by a trusted interpreter. **Chan event** handlers are *always* executed in the interpreter that set them up.

chan push *channelId cmdPrefix*

Adds a new transformation on top of the channel *channelId*. The *cmdPrefix* argument describes a list of one or more words which represent a handler that will be used to implement the transformation. The command prefix must provide the API described in the **transchan** manual page. The result of this subcommand is a handle to the transformation. Note that it is important to make sure that the transformation is capable of supporting the channel mode that it is used with or this can make the channel neither readable nor writable.

chan puts ?**-nonewline**? ?*channelId*? *string*

Writes *string* to the channel named *channelId* followed by a newline character. A trailing newline character is written unless the optional flag **-nonewline** is given. If *channelId* is omitted, the string is written to the standard output channel, **stdout**.

Newline characters in the output are translated by **chan puts** to platform-specific end-of-line sequences according to the currently configured value of the **-translation** option for the channel (for example, on PCs newlines are normally replaced with carriage-return-linefeed sequences; see **chan configure** above for details).

Tcl buffers output internally, so characters written with **chan puts** may not appear immediately on the output file or device; Tcl will normally delay output until the buffer is full or the channel is closed. You can force output to appear immediately with the **chan flush** command.

When the output buffer fills up, the **chan puts** command will normally block until all the buffered data has been accepted for output by the operating system. If *channelId* is in non-blocking mode then the **chan puts** command will not block even if the operating system cannot accept the data. Instead, Tcl continues to buffer the data and writes it in the background as fast as the underlying file or device can accept it. The application must use the Tcl event loop for non-blocking output to work; otherwise Tcl never finds out that the file or device is ready for more output data. It is possible for an arbitrarily large amount of data to be buffered for a channel in non-blocking mode, which could consume a large amount of memory. To avoid wasting memory, non-blocking I/O should normally be used in an event-driven fashion with the **chan event** command (do not invoke **chan puts** unless you have recently been notified via a file event that the channel is ready for more output data).

chan read *channelId* ?*numChars*?

chan read ?**-nonewline**? *channelId*

In the first form, the result will be the next *numChars* characters read from the channel named *channelId*; if *numChars* is omitted, all characters up to the point when the channel would signal a failure (whether an end-of-file, blocked or other error condition) are read. In the second form (i.e. when *numChars* has been omitted) the flag **-nonewline** may be given to indicate that any trailing newline in the string that has been read should be trimmed.

If *channelId* is in non-blocking mode, **chan read** may not read as many characters as requested: once all available input has been read, the command will return the data that is available rather than blocking for more input. If the channel is configured to use a multi-byte encoding, then there may actually be some bytes remaining in the internal buffers that do not form a complete character. These bytes will not be returned until a complete character is available or end-of-file is

reached. The **-nonewline** switch is ignored if the command returns before reaching the end of the file.

Chan read translates end-of-line sequences in the input into newline characters according to the **-translation** option for the channel (see **chan configure** above for a discussion on the ways in which **chan configure** will alter input).

When reading from a serial port, most applications should configure the serial port channel to be non-blocking, like this:

chan configure *channelId* **-blocking** *0*.

Then **chan read** behaves much like described above. Note that most serial ports are comparatively slow; it is entirely possible to get a **readable** event for each character read from them. Care must be taken when using **chan read** on blocking serial ports:

chan read *channelId numChars*
In this form **chan read** blocks until *numChars* have been received from the serial port.

chan read *channelId*
In this form **chan read** blocks until the reception of the end-of-file character, see **chan configure -eofchar**. If there no end-of-file character has been configured for the channel, then **chan read** will block forever.

chan seek *channelId offset* ?*origin*?
Sets the current access position within the underlying data stream for the channel named *channelId* to be *offset* bytes relative to *origin*. *Offset* must be an integer (which may be negative) and *origin* must be one of the following:

start
The new access position will be *offset* bytes from the start of the underlying file or device.

current
The new access position will be *offset* bytes from the current access position; a negative *offset* moves the access position backwards in the underlying file or device.

end
The new access position will be *offset* bytes from the end of the file or device. A negative *offset* places the access position before the end of file, and a positive *offset* places the access position after the end of file.

The *origin* argument defaults to **start**.

Chan seek flushes all buffered output for the channel before the command returns, even if the channel is in non-blocking mode. It also discards any buffered and unread input. This command returns an empty string. An error occurs if this command is applied to channels whose underlying file or device does not support seeking.

Note that *offset* values are byte offsets, not character offsets. Both **chan seek** and **chan tell** operate in terms of bytes, not characters, unlike **chan read**.

chan tell *channelId*
Returns a number giving the current access position within the underlying data stream for the channel named *channelId*. This value returned is a byte offset that can be passed to **chan seek** in order to set the channel to a particular position. Note that this value is in terms of bytes, not characters like **chan read**. The value returned is -1 for channels that do not support seeking.

chan truncate *channelId* ?*length*?
Sets the byte length of the underlying data stream for the channel named *channelId* to be *length* (or to the current byte offset within the underlying data stream if *length* is omitted). The channel is flushed before truncation.

EXAMPLES

This opens a file using a known encoding (CP1252, a very common encoding on Windows), searches for a string, rewrites that part, and truncates the file after a further two lines.

```
set f [open somefile.txt r+]
chan configure $f -encoding cp1252
set offset 0

# Search for string "FOOBAR" in the file
while {[chan gets $f line] >= 0} {
    set idx [string first FOOBAR $line]
    if {$idx > -1} {
        # Found it; rewrite line
```

```
        chan seek $f [expr {$offset + $idx}]
        chan puts -nonewline $f BARFOO

        # Skip to end of following line, and truncate
        chan gets $f
        chan gets $f
        chan truncate $f

        # Stop searching the file now
        break
    }

    # Save offset of start of next line for later
    set offset [chan tell $f]
}
chan close $f
```

A network server that does echoing of its input line-by-line without preventing servicing of other connections at the same time.

```
# This is a very simple logger...
proc log {message} {
    chan puts stdout $message
}

# This is called whenever a new client connects to the server
proc connect {chan host port} {
    set clientName [format <%s:%d> $host $port]
    log "connection from $clientName"
    chan configure $chan -blocking 0 -buffering line
    chan event $chan readable [list echoLine $chan $clientName]
}

# This is called whenever either at least one byte of input
# data is available, or the channel was closed by the client.
proc echoLine {chan clientName} {
    chan gets $chan line
    if {[chan eof $chan]} {
        log "finishing connection from $clientName"
        chan close $chan
    } elseif {![chan blocked $chan]} {
        # Didn't block waiting for end-of-line
        log "$clientName - $line"
        chan puts $chan $line
    }
}

# Create the server socket and enter the event-loop to wait
# for incoming connections...
socket -server connect 12345
vwait forever
```

SEE ALSO

close, **eof**, **fblocked**, **fconfigure**, **fcopy**, **file**, **fileevent**, **flush**, **gets**, **open**, **puts**, **read**, **seek**, **socket**, **tell**, **refchan**, **transchan**

KEYWORDS

channel, input, output, events, offset